Inverted double pendulum nonlinear estimation example

Ioannis Mandralis, 26 Feb 2023

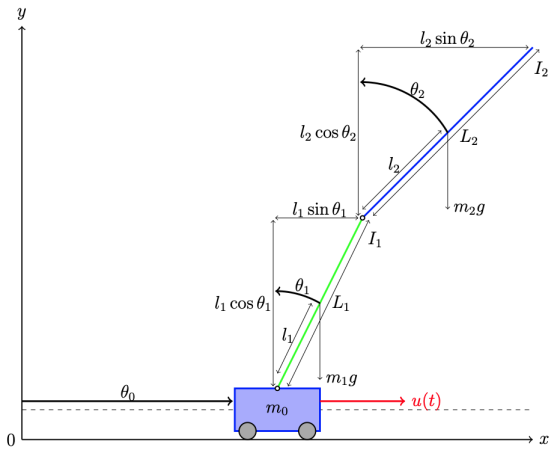
In this example we work through estimation of the state of the inverted double pendulum system with a sensor capable of measuring the full state. First we look at estimating the noisy state subject to process and measurement noise using a perfect model. Then we assume we have a slightly perturbed model and try to estimate the state again (in practical applications such a model might have been obtained using machine learning for example).

All calculations are done in discrete time, first using the Extended Kalman Filter in Predictor-Corrector form and then comparing to the Unscented Kalman Filter.

In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct

# Define some line styles for later use
ebarstyle = {'elinewidth': 0.5, 'capsize': 2}
xdstyle = {'color': 'k', 'linestyle': '--', 'linewidth': 0.5,
           'marker': '+', 'markersize': 4}
```

System description

The state of the inverted double pendulum is $(x, \theta_1, \theta_2, \dot{x}, \dot{\theta}_1, \dot{\theta}_2)$. The control input is the linear force applied to the cart u.



The dynamics of the inverted double pendulum are given by:

$$D(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = Hu$$

where $q = (x, \theta_1, \theta_2)$ is the generalized coordinate. This system has been transformed into a 6 state first order ODE and the matrices D, C, G, H have been implemented in the associated python file. Note: in the image above θ_0 is meant to correspond to x.

In [2]:
```
from dinvpend import dbpend_noisy, dbpend_noisy_perturbed, plot_results, plo
from ukf import UKF
print(dbpend_noisy)
```

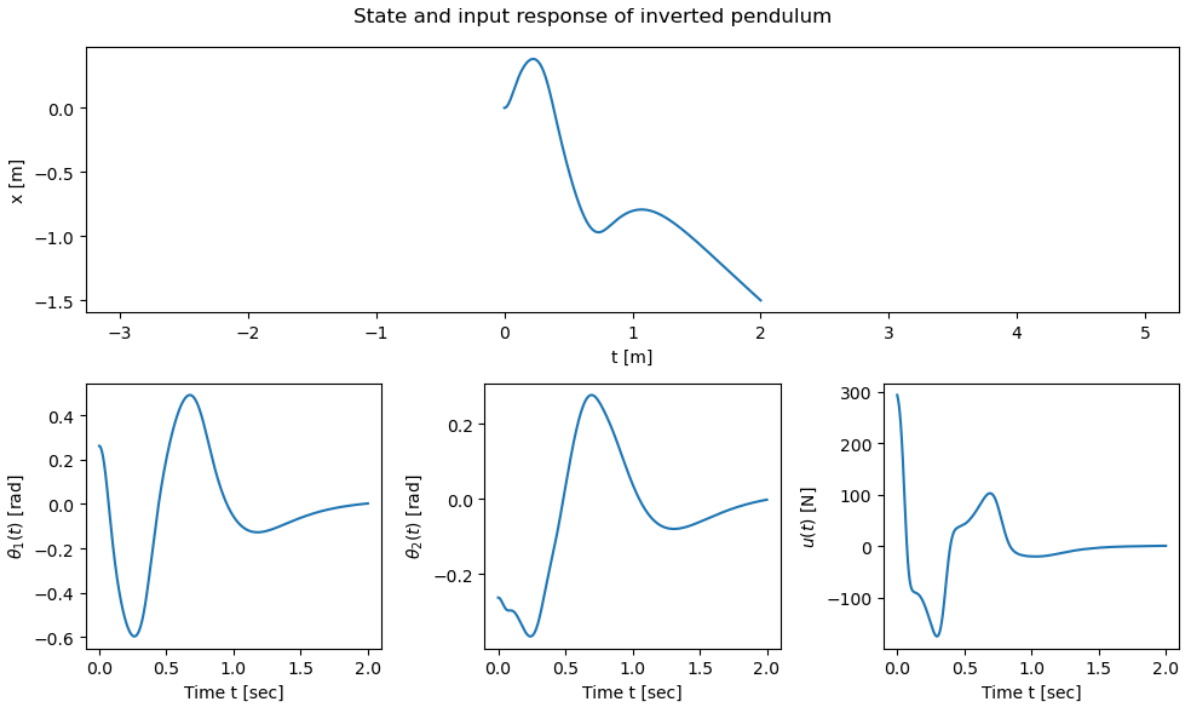
```
<NonlinearIOSystem>: dbpend
Inputs (4): ['u', 'vx', 'vth1', 'vth2']
Outputs (6): ['x', 'th1', 'th2', 'xdot', 'th1dot', 'th2dot']
States (6): ['x', 'th1', 'th2', 'xdot', 'th1dot', 'th2dot']

Update: <function dbpend_update_noisy at 0x7f9341779a20>
Output: <function dbpend_output at 0x7f934177add0>
```

In [3]:
```
# Find the equilibrium point corresponding to both links in the upright posi
xeq, ueq = np.zeros(6), np.zeros(4)
print(f"xeq:{xeq}")
print(f"ueq:{ueq}")

linsys = dbpend_noisy.linearize(xeq,ueq)
Q,R = np.diag([1,10,10,1,1,1]), 100
K,_,_ = ct.lqr(linsys.A,np.expand_dims(linsys.B[:,0],axis=1),Q,R)

def controller_output(t,x,z,params):
    xd = z[:6]
    ud = z[6:7]
    x  = z[7:]

    return ud + K @ (xd - x)

ctrl = ct.NonlinearIOSystem(
    None, controller_output, name='ctrl',
    inputs=['xd', 'th1d','th2d','xdotd','th1dotd','th2dotd','ud',
            'x','th1','th2','xdot','th1dot','th2dot'],
    outputs=['u']
)

# Create the complete control system
clsys = ct.interconnect((dbpend_noisy, ctrl), name='clsys',
        inputs=['xd','th1d','th2d','xdotd','th1dotd','th2dotd','ud','vx','vt
        outputs=['x','th1','th2','xdot','th1dot','th2dot','u']
)
```

```
xeq:[0. 0. 0. 0. 0. 0.]
ueq:[0. 0. 0. 0.]
```

In [4]:
```
# Simulate the closed loop system

# Start the pendulum in a "jackknife" position with no input and no velocity
x0 = (0.0,np.deg2rad(15),np.deg2rad(-15),0.0,0.0,0.0)
u0 = 0.0

# Plot the step response with respect to the reference input
Tf = 2.0
```

```
Ts = 0.0001
tvec = np.arange(0, Tf+Ts, Ts)
xd = xeq
ud = ueq[0]

# Add disturbances to the state
np.random.seed(117)
Rv = np.diag([10, 10, 10])*Ts
V = ct.white_noise(tvec, Rv, dt=Ts)

U = np.vstack([np.outer(xd,np.ones_like(tvec)), np.outer(ud,np.ones_like(tve
timepts, output = ct.input_output_response(clsys, tvec, U, X0=x0)
x,u = output[:6],output[6]
plot_results(timepts,x,u)

# let the desired trajectory be xd,ud
xd,ud = x,u
```

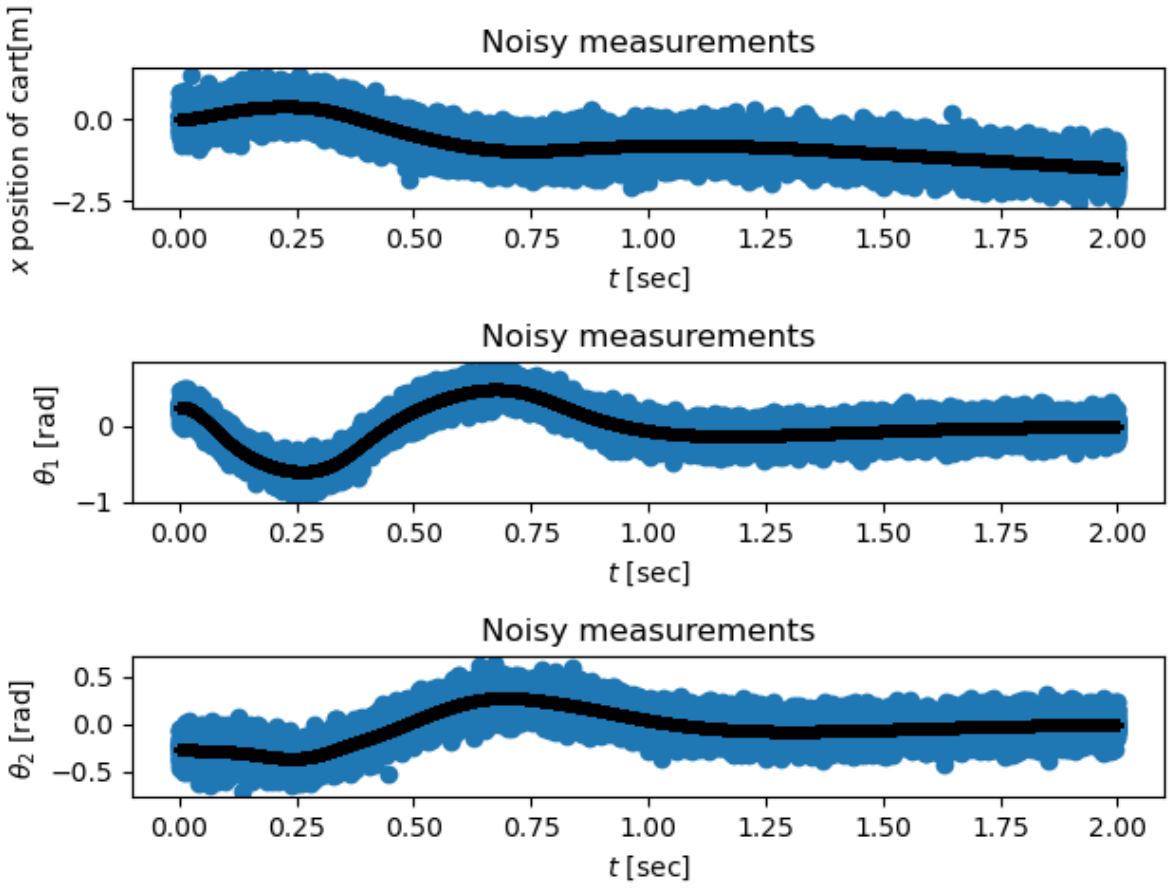


Sensor Model

First assume we can measure all states (subject to noise) and further assume that there is a disturbance entering in $\ddot{x}, \ddot{\theta}_1, \ddot{\theta}_2$.

In [5]:
```
# Disturbance and noise intensities
Rw = np.diag([0.1,0.01,0.01,0.1,0.01,0.01])

# Create process disturbance and sensor noise vectors
W = ct.white_noise(timepts, Rw, dt=Ts)

# Create the noisy inputs for our estimators by corrupting the state with no
# and also giving the commanded inputs
Y = xd + W
```

```
U = np.vstack([Y, ud])

# Plot the noisy output signal
plt.subplot(3, 1, 1)
plt.plot()
plt.scatter(timepts, Y[0])
plt.plot(timepts, xd[0], **xdstyle)
plt.xlabel("$t$ [sec]")
plt.ylabel(r"$x$ position of cart[m]")
plt.title("Noisy measurements")

plt.subplot(3, 1, 2)
plt.plot()
plt.scatter(timepts, Y[1])
plt.plot(timepts, xd[1], **xdstyle)
plt.xlabel("$t$ [sec]")
plt.ylabel(r"$\theta_1$ [rad]")
plt.title("Noisy measurements")

plt.subplot(3, 1, 3)
plt.plot()
plt.scatter(timepts, Y[2])
plt.plot(timepts, xd[2], **xdstyle)
plt.xlabel("$t$ [sec]")
plt.ylabel(r"$\theta_2$ [rad]")
plt.title("Noisy measurements")

plt.tight_layout()
```



Discrete Time System

In order to implement the Extended Kalman Filter in discrete time we need to first approximate the continuous dynamics by a discrete time system. Our system is of the form:

$$\dot{x} = f(x, u)$$

To convert to discrete time use the forward Euler method to approximate the discrete state dynamics

$$x[k+1] = x[k] + T_s f(x[k], v[k], u[k]) =: F(x[k], v[k], u[k])$$

F is defined below:

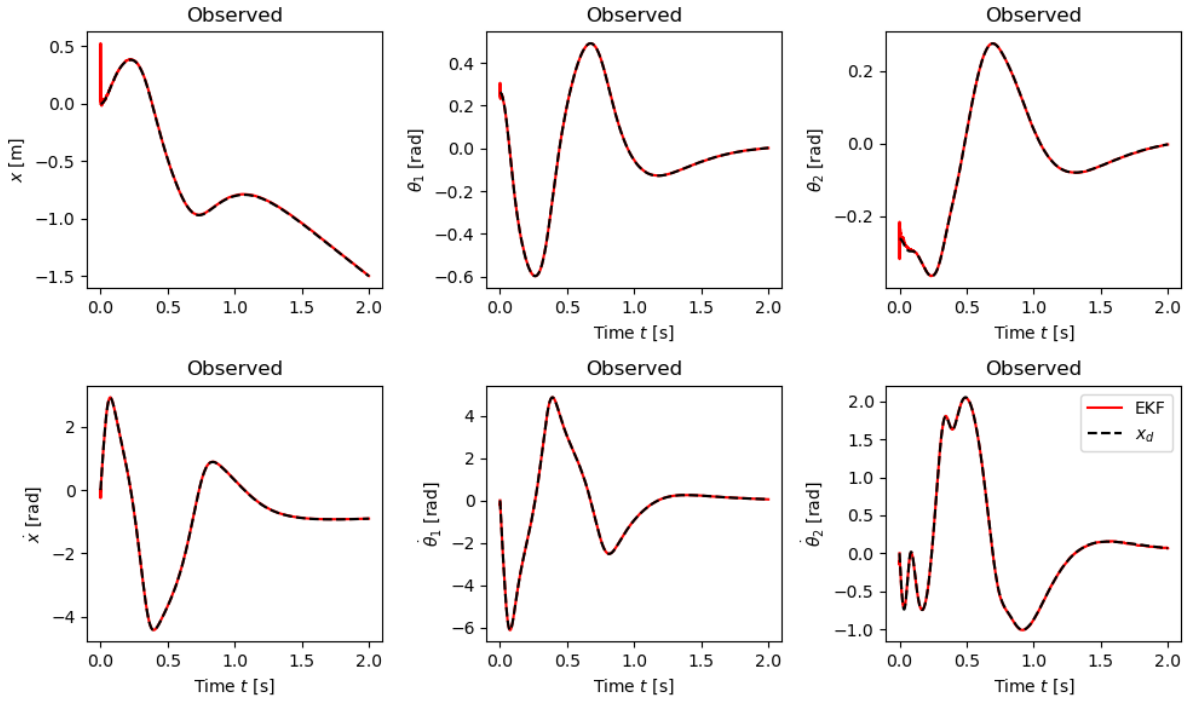
In [6]:
```
# Define the discrete time update function
def F(x, v, u):
    f_ = dbpend_noisy.updfcn(0.0,x,np.hstack([u,v]),{
        'L1': 0.5,              # length of link 1
        'L2': 0.5,              # length of link 2
        'm1': 1.0,              # mass of link 1
        'm2': 1.0,              # mass of link 2
        'm0': 4.0,              # mass of cart
        'g' : 9.81              # gravitational acceleration
    })
    return x + Ts*f_
```

Extended Kalman Filter

We now construct a discrete time Extended Kalman Filter to estimate the system state given the noisy measurements and the commanded control inputs.

In [7]:
```
# Define the disturbance input and measured output matrices
C = np.eye(6)

# Create an array to store the results
xhat = np.zeros((dbpend_noisy.nstates, timepts.size))
P = np.zeros((dbpend_noisy.nstates, dbpend_noisy.nstates, timepts.size))

# Update the estimates at each time
for i, t in enumerate(timepts):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:, 0]
        Pkkm1 = np.eye(dbpend_noisy.nstates)
    else:
        # Prediction step
        linsys = dbpend_noisy.linearize(xkk,np.hstack([ud[i-1],np.array([0,0
        A = np.eye(dbpend_noisy.nstates) + Ts*linsys.A
        F_ = Ts * linsys.B[:,1:]
```

```
        xkkm1 = F(xkk,np.array([0,0,0]),np.array([ud[i-1]]))
        Pkkm1 = A @ Pkk @ A.T + F_ @ Rv @ F_.T

    # Correction step
    L = Pkkm1 @ C.T @ np.linalg.inv(Rw + C @ Pkkm1 @ C.T)
    xkk = xkkm1 - L @ (C @ xkkm1 - Y[:, i])
    Pkk = Pkkm1 - L @ C @ Pkkm1

    # Save the state estimate and covariance for later plotting
    xhat[:, i], P[:, :, i] = xkkm1, Pkkm1  # For comparison to Kalman form

plt.figure(figsize=(10,6))
plot_estimate(timepts,xhat,P,xd,'EKF','r',obsv=[1,1,1,1,1,1])
plt.legend()
plt.tight_layout()
```



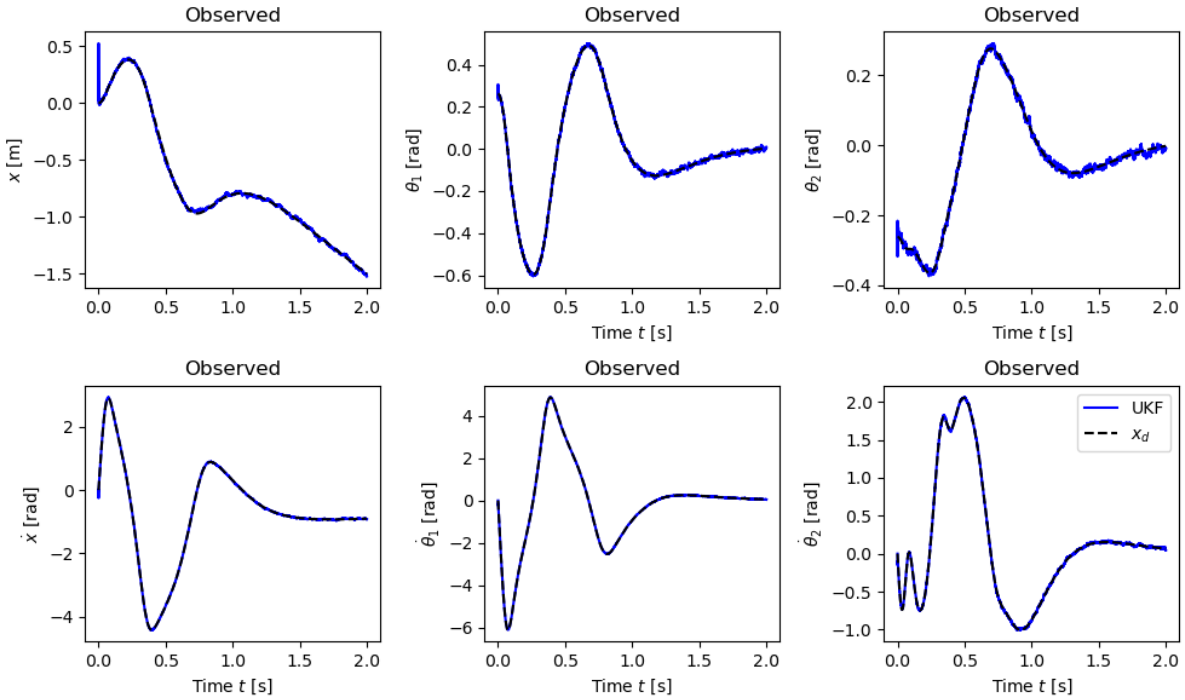
Unscented Kalman Filter

We now construct an Unscented Kalman Filter for the system.

In [8]:
```
# Define the output function
def H(x,w):
    return C@x+w

# Initialize the unscented kalman filter
ukf = UKF(dim_x=6, dim_z=6, Q=Rv, R=Rw, kappa=0.0)

# Create an array to store the results
xhat_ukf = np.zeros((dbpend_noisy.nstates, timepts.size))
P_ukf = np.zeros((dbpend_noisy.nstates, dbpend_noisy.nstates, timepts.size))

# Update the estimates at each time
```

```
for i, t in enumerate(timepts):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:,0]
        Pkkm1 = np.eye(dbpend_noisy.nstates)
    else:
        # Prediction step
        xkkm1, Pkkm1, _ = ukf.predict(F, xkk, Pkk, np.array([ud[i-1]]))

    # Correction step
    xkk, Pkk, _ = ukf.correct(H, xkkm1, Pkkm1, Y[:, i])

    # Save the state estimate and covariance for later plotting
    xhat_ukf[:, i], P_ukf[:, :, i] = xkkm1, Pkkm1

plt.figure(figsize=(10,6))
plot_estimate(timepts,xhat_ukf,P_ukf,xd,'UKF','b',obsv=[1,1,1,1,1,1])
# plot_estimate(timepts,xhat,P,xd,'EKF','r',obsv=[1,1,1,1,1,1])
plt.legend()
plt.tight_layout()
```

```
W0m:0.0
W0c:0.0
Wi:0.03333333333333333
```



Perturbed model

Now we try to estimate the system with a model which is slightly wrong

In [9]:

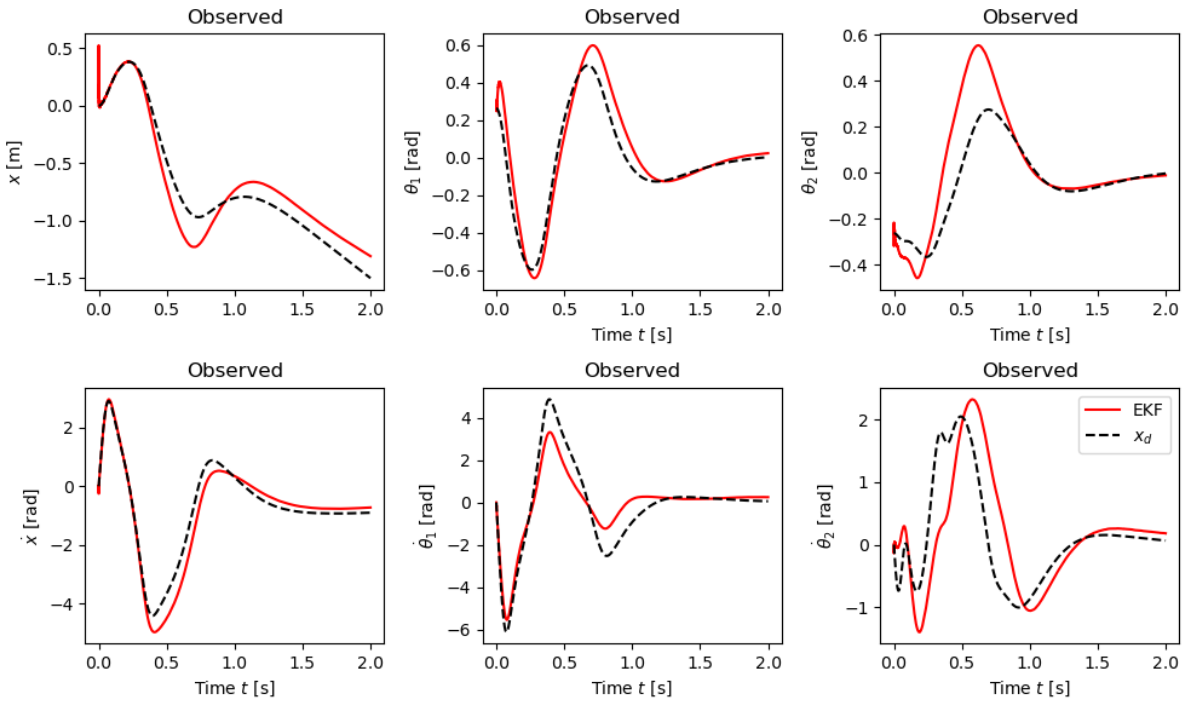
```
from dinvpend import dbpend_noisy_perturbed

# Define the discrete time update function
```

```
def F(x, v, u):
    f_ = dbpend_noisy_perturbed.updfcn(0.0,x,np.hstack([u,v]),{
        'L1': 0.5,              # length of link 1
        'L2': 0.5,              # length of link 2
        'm1': 1.0,              # mass of link 1
        'm2': 1.0,              # mass of link 2
        'm0': 4.0,              # mass of cart
        'g' : 9.81              # gravitational acceleration
    })
    return x + Ts*f_
```

Extended Kalman Filter with perturbed model

We now construct a discrete time Extended Kalman Filter to estimate the system state given the noisy measurements and the commanded control inputs using the perturbed model.

In [10]:
```
# Define the disturbance input and measured output matrices
C = np.eye(6)

# Create an array to store the results
xhat = np.zeros((dbpend_noisy_perturbed.nstates, timepts.size))
P = np.zeros((dbpend_noisy_perturbed.nstates, dbpend_noisy_perturbed.nstates

# Update the estimates at each time
for i, t in enumerate(timepts):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:, 0]
        Pkkm1 = np.eye(dbpend_noisy_perturbed.nstates)
    else:
        # Prediction step
        linsys = dbpend_noisy_perturbed.linearize(xkk,np.hstack([ud[i-1],np.
        A = np.eye(dbpend_noisy_perturbed.nstates) + Ts*linsys.A
        F_ = Ts * linsys.B[:,1:]
        xkkm1 = F(xkk,np.array([0,0,0]),np.array([ud[i-1]]))
        Pkkm1 = A @ Pkk @ A.T + F_ @ Rv @ F_.T

    # Correction step
    L = Pkkm1 @ C.T @ np.linalg.inv(Rw + C @ Pkkm1 @ C.T)
    xkk = xkkm1 - L @ (C @ xkkm1 - Y[:, i])
    Pkk = Pkkm1 - L @ C @ Pkkm1

    # Save the state estimate and covariance for later plotting
    xhat[:, i], P[:, :, i] = xkkm1, Pkkm1  # For comparison to Kalman form

plt.figure(figsize=(10,6))
plot_estimate(timepts,xhat,P,xd,'EKF','r',obsv=[1,1,1,1,1,1])
plt.legend()
plt.tight_layout()
```

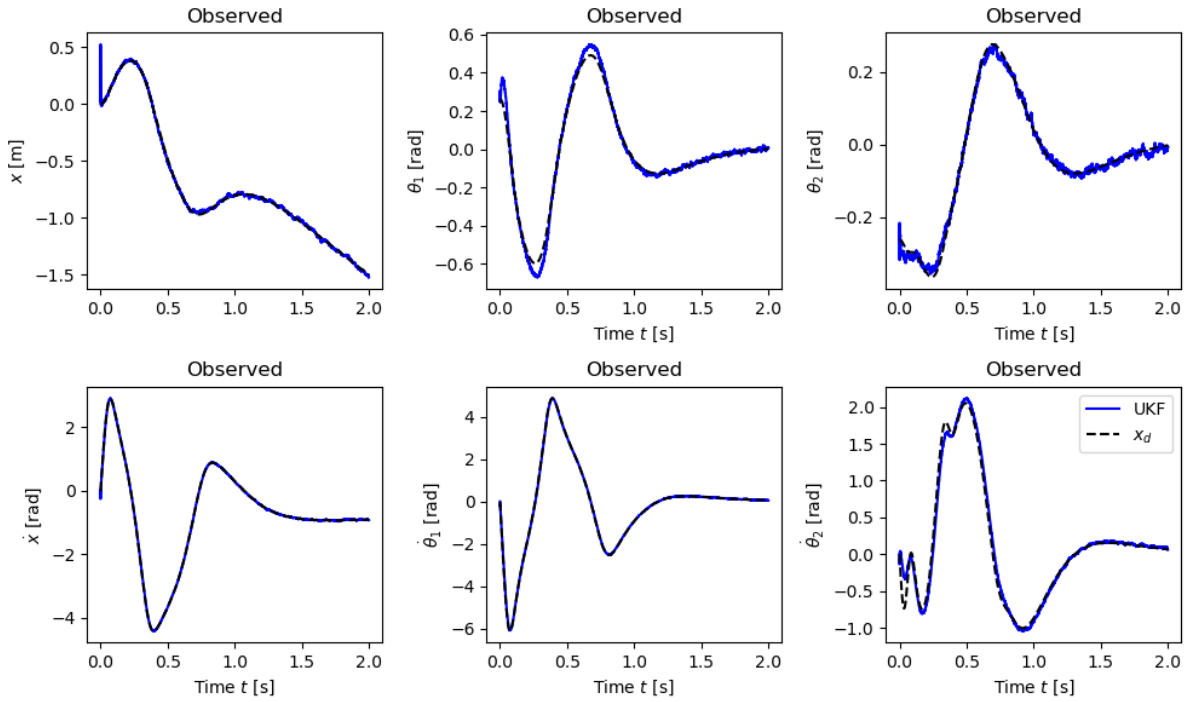


Unscented Kalman Filter with perturbed model

In [11]:

```
# Initialize the unscented kalman filter
ukf = UKF(dim_x=6, dim_z=6, Q=Rv, R=Rw, kappa=0.0)

# Create an array to store the results
xhat_ukf = np.zeros((dbpend_noisy_perturbed.nstates, timepts.size))
P_ukf = np.zeros((dbpend_noisy_perturbed.nstates, dbpend_noisy_perturbed.nst

# Update the estimates at each time
for i, t in enumerate(timepts):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:,0]
        Pkkm1 = np.eye(dbpend_noisy_perturbed.nstates)
    else:
        # Prediction step
        xkkm1, Pkkm1, _ = ukf.predict(F, xkk, Pkk, np.array([ud[i-1]]))

    # Correction step
    xkk, Pkk, _ = ukf.correct(H, xkkm1, Pkkm1, Y[:, i])

    # Save the state estimate and covariance for later plotting
    xhat_ukf[:, i], P_ukf[:, :, i] = xkkm1, Pkkm1

plt.figure(figsize=(10,6))
plot_estimate(timepts,xhat_ukf,P_ukf,xd,'UKF','b',obsv=[1,1,1,1,1,1])
# plot_estimate(timepts,xhat,P,xd,'EKF','r',obsv=[1,1,1,1,1,1])
plt.legend()
plt.tight_layout()
plt.show()
```

```
W0m:0.0
W0c:0.0
Wi:0.03333333333333333
```



In [ ]: